Parallel adaptive mesh refinement for incompressible flow problems

R. Rossi[1,2], J. Cotela[1,2], N. M. Lafontaine[2] , P. Dadvand[1,2] and S. Idelsohn[1,3]
[1] Centre Internacional de Mètodes Numèrics en Enginyeria (CIMNE).
Gran Capità s/n, Edifici C1, 08034 Barcelona, Spain.
[2] UPC BarcelonaTech, Campus Nord UPC, 08034 Barcelona, Spain.
[3] ICREA Research Professor
email: rrossi@cimne.upc.edu

January 30, 2012

Abstract

The present article describes a simple element-driven strategy for the conforming refinement of simplicial finite element meshes in a distributed environment. The proposed algorithm is effective both for local adaptive refinement and for the division of all the elements within an existing mesh. We aim to provide sufficient detail to allow the practical implementation of the algorithm, which can be coded with minimal effort provided that a distributed linear algebra library is available. The proposed refinement strategy is composed of three basic components: a global splitting strategy, an elemental splitting procedure and an error estimation technique, which are combined so to guarantee obtaining a conformant refined mesh. A number of benchmark examples show the capabilities of the proposed method. Error is estimated for the incompressible fluid-flow benchmarks using a novel indicator based on the computation of the sub-scale velocity.

Keywords: adaptive mesh refinement, parallel computing, incompressible Navier-Stokes, subgrid error estimation.

1 Introduction

The importance of refining a mesh in order to capture increasingly fine details is well known in the numerical community, while the possibility of providing an increased level of refinement in selected areas of the domain constitutes the key to the success of non-structured discretizations over structured alternatives. The use of Adaptive Mesh Refinement (AMR) techniques in order to improve adaptively the resolution during the solution process, was soon detected as a very attractive possibility and was used extensively in different areas of engineering, but particularly in the field of compressible CFD. Mesh refinement is typically less computationally expensive than remeshing, as it uses the existing connectivity as a starting point, which simplifies the mapping of the data associated to the original nodes and elements to the new ones. In particular, if the refinement is performed by adding new nodes along the edges of the original elements, as in the case of the algorithm presented here, interpolating data to the new mesh is a trivial operation.

A number of different algorithms can be found in the literature describing multiple approaches to the splitting of simplicial elements. Most techniques are based on some kind of edge-subdivision, which has proved to be very effective and relatively easy to code in a serial context. Despite the existence of various successful implementations, it is still generally difficult to port the original serial algorithms to a distributed environment.

An early investigation of local mesh refinement can be found in the works of Babuscka, Rheinboldt et al. in the FEARS project [3]. Since then, local and global mesh refinement and coarsening of triangular and tetrahedral meshes have been used for finite element methods for solving linear elliptic partial differential equations [5] , optimization and visualization of volume data [16], magnetostatic problems [33], compressible flows [34], finite element computation of magnetic fields and electromagnetic problems [17, 19, 26, 38], and generate a nearly optimal mesh in which the discretization error is equally distributed with the help of an error estimation [8, 15].

Some commonly used refinement approaches are the regular refinement, the bisection [3, 13, 17, 19, 28, 30, 44], and refinement using the Delaunay criterion and edge splitting.

The regular refinement consists in simultaneously bisecting all edges of the triangle or tetrahedron to be refined, producing 4 smaller triangles or 8 smaller tetrahedra. Since the regular refinement cannot generate locally refined conforming meshes, either special numerical algorithms are designed to handle the hanging nodes, or it is combined with other types of refinement to produce a conforming mesh. On the other hand, with the bisection refinement, only one edge of the triangle or the tetrahedron, called the refinement edge, is bisected, producing 2 smaller triangles or tetrahedra. The main advantage of bisection refinement is that it automatically produces locally refined conforming meshes and nested finite element spaces. A major drawback of the simple edge bisection algorithm is the poor mesh quality, most notably after several refinements. The main problem with bisecting refinement is how to select the refinement edge such that triangles or tetrahedra produced by successive refinements do not degenerate.

The methods for selecting the refinement edge proposed by various authors can be classified into two categories, namely the longest edge approach [21] and the newest vertex approach (the latter is also called the newest node approach by some authors).

A class of algorithms based on the refinement of the longest edge is proposed and studied by Rivara et al. [41]. In these algorithms, triangles or tetrahedra are always bisected using one of their longest edges, and the finite termination of the refinement procedure is obvious because when traversing from one simplex to another in order to make the resulting mesh conforming, one steps along paths of simplices with longer longest edges. In [42], some mathematical guarantees are provided on the quality of the resulting mesh.

The newest vertex approach was first proposed for two dimensional triangular meshes by Sewell [43], and was generalized to three dimensions by Bansch [6]. More recent work on the newest vertex algorithms is described in the papers of Kossaczky [29] and Liu and Joe [32]. The concept of the newest vertex approach is very simple in two dimensions: once the refinement edge for the triangles in the initial mesh is determined, the refinement edge of a newly generated triangle is the edge opposite to its newest vertex. Unfortunately, its generalization to three dimensions is highly non-trivial, and the algorithms proposed by various authors are essentially equivalent, but use different interpretations. It is theoretically proved that tetrahedra generated by these algorithms belong to a finite number of similarity cases, which ensures non-degeneracy of tetrahedra produced by repeated bisections.

There are also works in the literature on parallel mesh refinement algorithm for triangular and tetrahedra meshes [4, 13, 21, 28, 31, 31] and some of them use bisection schemes [44] . Rivara et al proposed a parallel algorithm for global refinement of tetrahedral meshes which is not suitable for adaptive local refinement. Pebay and Thompson [39] presented a parallel refinement algorithm for tetrahedral meshes based on edge splitting. Jones and Plassmann [27] proposed and studied a parallel algorithm for adaptive local refinement of two dimensional triangular meshes. Barry, Jones, and Plassmann [7] also presented a framework for implementing parallel adaptive finite element applications and demonstrated it with 2D and 3D plasticity problems. Lin-Bo Zang [44] presents a parallel algorithm for distributed memory parallel computers using bisection, which is characterized by allowing simultaneous refinement of submeshes to arbitrary levels before synchronization between submeshes and without the need of a central coordinator process for managing new vertices. His algorithm is based on the standard message passing interface MPI. The mesh is partitioned in submeshes, as many as number the of MPI processes. Partitioning is computed using METIS. After partitioning the mesh, in the first step of his algorithm, the submeshes are refined independently, with the shared faces treated as if they were boundary faces. Then, tetrahedra containing one or more shared faces which have been bisected during the first phase are exchanged between neighbor submeshes, and tetrahedra having one or more hanging shared faces are bisected. The process stops when global conformity of the mesh is reached, as the first step creates submeshes with non-conforming shared faces. H. L. De Cougny and M. S. Shephard [13] use edge-based subdivision templates for refinement, that is, they uses pre-defined templates to refine mesh regions. Like the other method, a two step process is necessary for parallel refinement: the first phase consists in subdividing mesh faces on partitions and then, on a second phase, the meshes are subdivided using the templates as in serial. However, one has to make sure that duplicate faces on partition boundaries are meshed identically. An improved simple cell-quality control for a large-scale unstructured tetrahedral mesh for parallel AMR was proposed by Y.-Y. Lian et al. [31]. It was designed such that the resulting refined mesh information can be readily utilized in both node or cell-based numerical methods Parallel implementation of its mesh refinement is widely discussed in the same reference.

Other mesh refinement strategies in the literature use a data structure for adaptive Finite Element computation of 2D and 3D problems. Using a hierarchical minimal tree based data structure for mesh refinement is proposed in [9]. This algorithm is implemented by imposing a one-level rule and using the adjacent neighbor (sharing edges

and faces) concept for recursive refinement. This technique generates mesh refinement data such as a connectivity matrix, an automatic local and global node numbering, a natural order of element sequence, and a coordinate array for the refined elements. A local mesh refinement using Local Delaunay Subdivisions is presented by T. W. Nehl and D. A. Field for solving magnetostatic problems [9].

Besides the need of organizing operations so to minimize "random" communications, the implementation of such techniques generally requires optimized custom data structures which are normally not easily available to code developers in other fields.

The present document describes a new strategy to perform local mesh refinement, based on the division of chosen elements by splitting their edges. The rationale at the base of our approach is to provide an algorithm that can be implemented with minimal effort, leveraging existing linear-algebra data structures. As we shall describe in the following, our technique assumes the availability of a distributed sparse-matrix package, preferably prepared for Finite Element assembly. Our testing was done using the Epetra package of the Trilinos Framework [20], but similar capability can be found in any other equivalent library. The only other dependency is a routine that provides the splitting pattern for all of the cases of interest. Our aim is to provide, together with this work, a liberally licensed version of such routine which considers all of the 729 (3^6) cases which may appear during the subdivision.

As a matter of fact a number of common steps are needed to allow the use of adaptive refinements within a distributed code, namely

- Implementation of a refinement strategy
- Definition of refinement indicators
- Preparation of the code to allow variations in the connectivity
- Load Balancing

While the last three of such points depend on the specific problem of interest as well as on the implementation of the software, the refinement strategy is "modular" with respect to the others. The objective of the present article is exactly to focus on such aspect, describing an algorithm that is amenable to a modular implementation within any unstructured distributed code (on simplicial meshes). In accordance with this goal, emphasis will be placed on the more practical aspects of the algorithm and its implementation on a distributed environment, and mesh quality will not be considered in this document.

In addition, an error estimator for incompressible flow problems will be presented in the final part of this document. This estimator will allow the definition of a refinement criterion which will be used in the application examples. Obviously, other techniques will be required to determine where refinement should be performed in other fields but, as mentioned, this last component is independent of the refinement algorithm.

2 Global Splitting Algorithm

In current section, we aim to describe our proposal for the subdivision algorithm. The goal is to obtain a conformant mesh by splitting an arbitrarily defined subset of the elements in the mesh (which may well contain the totality of the elements).

Our approach is based in the division of the elements identified as candidates for refinement by introducing new nodes on the midpoints of their edges, dividing each triangle in four smaller ones or each tetrahedron in eight parts. Adjacent elements are split accordingly to preserve a conforming connectivity.

The approach is designed under the rather standard assumption that the domain splitting assigns each tetrahedron univocally to one of the MPI domains. Here, nodes are assigned to a single process, which is said to "own" them although, occasionally, some of these nodes will appear in elements owned by a different process. Nodes which appear on a process' local elements but are not owned by it are referred throughout this document as "ghost" nodes. It is assumed that all processes will store a copy of all their "ghost" nodes and the database of nodal values associated to them.

Each node is expected to have a uniquely defined global Id (GID) and to store the rank of the owner process, and each edge is univocally identified by the GIDs of its end nodes.

Since the algorithm is rather complex and articulated in six steps, we believe that the only viable presentation is by providing a commented pseudo-code. In order to abstract from a specific implementation of the linear algebra library, we will assume that some capabilities are provided by the linear algebra library, namely:

- `ConstructLocal_FEGraph`

  Function that defines the "local connectivity", that is, the connectivity of all of the nodes that are needed within a single MPI process and without communications (this will include both owned and ghost nodes for a given MPI process). This abstract function should take as input a given finite element submesh, that is, the list of elements of the corresponding subdomain, and construct the corresponding graph.

- `ConstructByLocal_Graph`

  Function that allows assembling the global graph of a matrix given the local graphs of which it is made, taking care of all of the MPI communications needed.

- `copy`

  Function that creates a copy of a global matrix. Note that this can be done without communication, locally in each domain.

- `SetScalarValue`

  Function to set to a given scalar number all of the non-zero entries in a matrix. Note that no communication is required to do this.

- `ADD_LocalMatrix`

  Function that *assembles* a local (sparse) matrix into the global one, by summing up local contributions into the corresponding global positions, taking care of all of the MPI communications needed.

- `REPLACE_LocalMatrix`

  Function that *replaces* the values contained in a local (sparse) matrix into the global one, taking care of all of the MPI communications needed. Note that the results of this operation are not required to be deterministic, in the sense that they could be allowed to vary between runs.

- `GetLocalView`

  Function that allows obtaining a local view of the terms of the global matrix which are described by the local graph. This function is expected to handle all of the MPI communications needed for importing matrix elements that are stored remotely to a locally available sparse matrix.

Such functions map naturally to the implementation provided by the Trilinos Epetra package (the REPLACE function requires a feature first introduced in Trilinos version 10.8.3), but they could be implemented with relative ease within the framework provided by other distributed packages. The key idea, is that the synchronization will be hidden by the interface between local and global matrices, so that by obtaining a "local view" of a global matrix, we will have access to the synchronized values after the different local contributions were added or replaced within the global matrix.

We should also remark that the matrix to be used is expected to be symmetric, so that each edge with GIDs (I,J), is biunivocally associated to a single matrix entry. Symmetry is in particular crucial if a non-deterministic implementation of the REPLACE function is used.

On the base of such definitions we can thus describe the steps of our algorithm, which can be written as

1. First of all we need to construct the local graph, intended as a sparse matrix that has a non zero entry for any couple (I,J) of indices that identify an edge. Such matrix should be constructed in an elementwise fashion so that any edge in the local FE mesh will be automatically part of the matrix.

   The local connectivity will be then used to allocate global matrices which will be stored in the "natural" format of the underlying linear algebra library.

   In this step we will also allocate the memory for all of the sparse matrices to be used in the implementation, namely "A", which we will use to identify the edges to be refined, "P" which will ultimately contain the "rank" to be assigned to each of the refinement nodes, that is, the parallel process that will own the node, and "IdMatrix" which will be used towards the end of the algorithm to store the GIDs of the new nodes.

```
my_rank = my MPI rank

ConstructLocal_FEGraph(local_graph)

A.ConstructByLocal_Graph(local_graph)
A.SetScalarValue(-1)

P = copy(A)
P.SetScalarValue(-1)

IdMatrix = copy(A)
IdMatrix.SetScalarValue(0)

Alocal = GetLocalView(A,local_graph)
Plocal = GetLocalView(P,local_graph)
```

2. As a second step we identify the elements that are scheduled for refinement by looping over the elements and following the arbitrary suggestions of the user. Note that the description of our refinement algorithm is completely independent on the approach used in choosing the elements to be refined. The decision of which elements to refine is left to the user, allowing for arbitrary refinement or, in the case of AMR, refinement based on the suggestions of an external error estimation routine.

   In our proposal, elements marked for refinement will be split by all their edges, resulting in four triangles or eight tetrahedra for each original element, a division that preserves the quality of the refined elements. Elements that share one or more edges with elements marked for refinement will be refined accordingly to preserve the connectivity, although in this case there is no guarantee on the final mesh quality. Of course different refinement strategies could be used at this point without changing the global algorithm.

   To identify the edges that will be refined, we have adopted the convention of adding -1 to the corresponding matrix entry in A. This identification of the edges to be refined will be performed first at local level and then communications will be performed to ensure a consistent behavior across processor boundaries. A local copy of the splitting pattern will be finally gathered for usage in the next steps.

```
loop on local elements
   if user requests splitting
      loop on edges of the element
          I,J = GID of the edges of the element
          Alocal(I,J) = Alocal(I,J) - 1

A.ADD_LocalMatrix(Alocal)
Alocal = GetLocalView(A,local_graph)
```

3. In the third step we assign the "owner" of the new nodes to be created. Here all of the MPI processes that will need a local copy of the node in edge I,J "propose" to be the owner by marking the corresponding values in Plocal with their MPI rank. The final owner of the node is not important as long as it is uniquely defined and known to all of the processes. We thus advocate the use of a "Replace" functions which guarantees that global values are overwritten by local contributions so that only the last contribution remains. This is non deterministic, but appears to work satisfactorily.

```
loop on local elements
   loop on edges of the element
       I,J = GIDs of the nodes at each element edge
       if(Alocal(I,J) < -1)
          Plocal(I,J) = my_rank

P.REPLACE_LocalMatrix(Plocal)

Plocal = GetLocalView(P,local_graph)
```

4. In the fourth step we know exactly which edges have to be created and who will be their owner. We need to find a suitable global Id, so that the global ids will be numbered consecutively, preferably avoiding gaps, and ensuring that no global id is repeated. The technique we propose is to have each of the different processes to count the new nodes of which it will be the owner, and to use a scan-sum based approach so that the new nodes will start from the greatest id found prior to splitting and have increasingly high id depending on their processor and position within the local nodes.

   Since all of the processors involved may need access to the id of the new ghost nodes, we will write the newly assigned id to a matrix and synchronize it between the processors. The pseudo-code of our proposal is thus something of the type:

```
local_node_counter = 0
for I in rows of Alocal
    for J in rows of Alocal
         if(Alocal(I,J) < -1)  //need remesh!
            if(Plocal(I,J) == my_rank) //we own the node
                local_node_counter= local_node_counter+1

use scan sum to determine new_id_local_start

new_id = new_id_local_start
for I in rows of Alocal
    for J in rows of Alocal
        if(Alocal(I,J) < -1) //need remesh!
             if(Plocal(I,J) == my_rank) //we own the node
                IdMatrix_local(I,J) = new_id
                new_id = new_id + 1

IdMatrix.ADD_LocalMatrix(IdMatrix_local)
IdMatrix_local = GetLocalView(IdMatrix,local_graph)
```

5. In the fifth step all the information needed to perform the creation of new nodes on the edges is already available locally. We will thus create them with the global id we identified, perform the interpolation of coordinates and nodal data from the edge vertices, and assign the "owner"

```
// admissible node Ids start with 1 (not zero!)
for I in rows of IdMatrix_local
    for J in rows of IdMatrix_local
       if(IdMatrix_local(I,J) > 0)
           generate new node in the middle of edge I,J
           interpolate from I and J
           assign global id =  IdMatrix_local(I,J)
           assigin node_rank = Plocal(I,J)
```

6. In the sixth step we generate the new elements using the newly created nodes. The difficulty (which we will address in next section) is to guarantee that the split mesh is still conformant.

   In order to do so we loop over the elements and, for each element, we gather the data associated to each of the edges. If a new node is associated to any of the edges, local refinement is needed and will be performed

   This is guaranteed by the helper functions we provide which we will discuss later on. We should remark that the creation of new nodes may be needed at this stage depending on the splitting pattern to be used. Nevertheless, eventual new nodes are purely local and therefore no new communication will be needed, besides the minimal one needed for assigning an unique global id.

```
loop on elements
   if any elemental edge is splitted
       gather newly created nodes
```

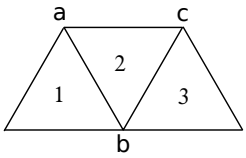


Figure 1: An example of a mesh where recalculation of the communication pattern will be required.

```
split element (see next section)
interpolate elemental data
```

The proposed algorithm has the advantage of being largely independent on the specific data structures used, and can be introduced with relative ease within a wide range of computational codes. A working implementation of the algorithm can be freely downloaded, together with the *Kratos Multiphysics* code (see [14]) at [1].

Before proceeding, we should remark that, once the algorithm finishes, the load can be severely unbalanced and action may be needed in this sense. Since the implementation of the load balancing step largely depends on the specific features of the different computational codes, we do not intend to discuss here the issue. In addition to this, the proposed renumbering scheme is not optimal and the users may wish to perform a renumbering pass in order to improve the ordering of nodes.

We should also observe that at the end of the refinement step, the communication pattern between the different domains might change. A simple example is needed to explain why this may happen. Let's consider the configuration shown in Fig(1). Assume that the original mesh, composed by triangles 1, 2 and 3 is partitioned so that these triangles are owned by processes 1, 2 and 3 respectively. Process 1 owns node a, while process 3 owns nodes b and c. No node is owned by processor 2.

In order to perform assembly operations, it is customary to gather nodal data to the owner which will sum the different contributions and finally spread it to the other nodes. For the configuration described, processor 1 needs to gather the data related to node a from processor 2 and the data of node b from processor 2 and 3. Processor 2 on the other hand will not need to gather any nodal data since it is not "owner" of any node. This implies that when coloring the communications node 2 will not be considered for the gathering phase.

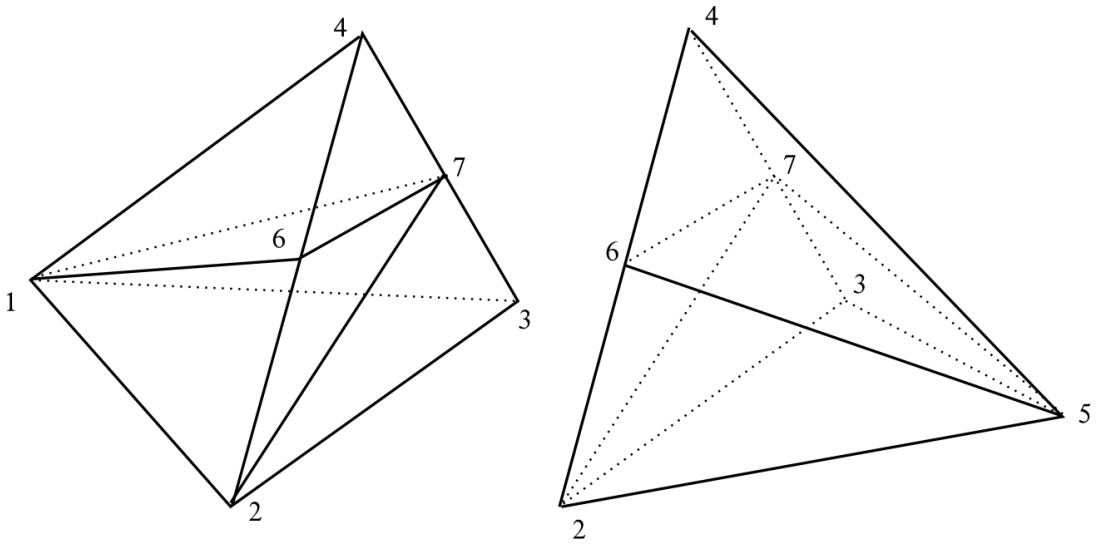
Let's suppose now that element 2 is marked for refinement and a new node has to be inserted on edge ab. With our algorithm, the owner of the new node can be either process 1 or 2. We will assume for the sake of the discussion that the owner is 2. If this happens, processor 2 becomes owner of a node and should be included in the gathering phase, hence invalidating the original coloring. As a consequence, the communication pattern has to be recomputed to take in account potential variations of the node ownerships. In general, the recalculation of an appropriate communication pattern between the different domains is implementation-dependent and falls aside of the objective of the current paper, nevertheless it is important to take in account that action may be needed to take care of this situation. Finally, on open problem is the optimization of the quality of the refined meshes. One possibility to be explored in this sense could be the use of ideas taken from "longest-edge" refinement algorithms in order to improve the quality of the refined meshes.

3 Splitting of simplicial elements

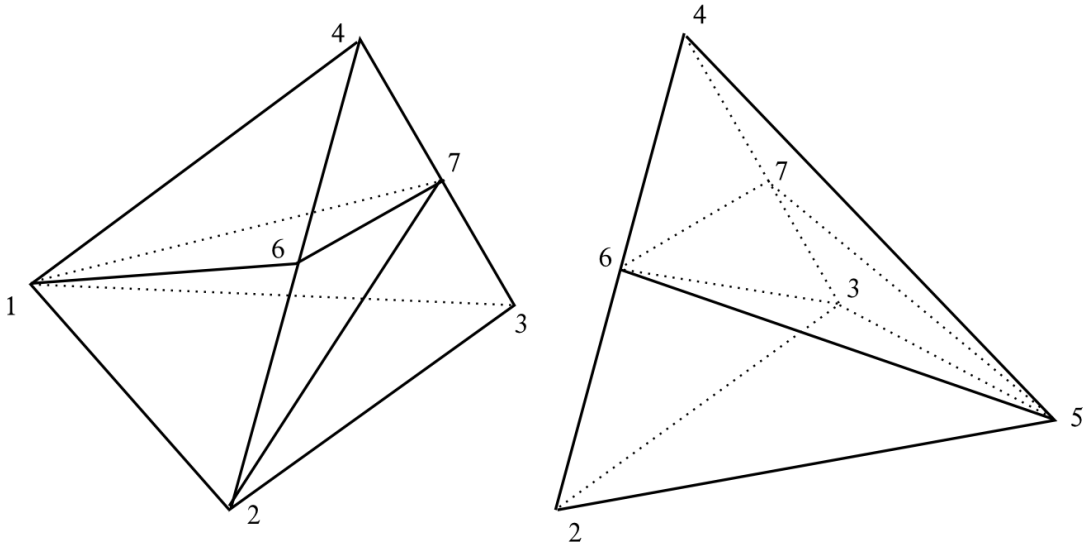
As observed in the introduction, the proposed implementation relies on the availability of both a flexible linear-algebra library and of an efficient splitting procedure. The aim of current section is to describe our implementation of the elemental splitting process, discussing briefly how to choose the splitting mode in order to guarantee obtaining a conformant mesh.

The basic idea is that the information available during the 6th step of the refinement algorithm (GID of the new nodes, and their owner) has to be sufficient to univocally define the splitting of the elements under the additional constraint that all the faces of a given tetrahedron should coincide with the faces of the neighboring element, that is to say, the refinement should be conformant.

This is best understood by a practical example: let us consider the two tetrahedra shown on the top part of Fig(2). The figure shows two original tetrahedra, with GIDs 1,2,3,4 and 2,5,3,4 respectively which have to be split by adding the new nodes 6 and 7. In the top part of the figure, a correct splitting is achieved as the edge (2,7) exists in both the tetrahedra that share the face 2 3 4. In the bottom part, on the other hand, the nodes 6 and 7 are



(a) Correct (conformant) splitting.



(b) Wrong (non-conformant) splitting.

Figure 2: Conformant and non-conformant splitting of two neighboring tetrahedra.

correctly added to both tetrahedra, but the the rightmost tetrahedra the edge (3,6) is included in the refined mesh while the edge (2,7) is created in the neighboring tetrahedra. As a consequence, the splitting is to be considered wrong as the face 2 3 4 is slitted differently in the two neighboring tetrahedra, leading to a non-conformant mesh.

Such potentially conflictive case happens when two nodes are added to two of the edges that form a face while the third edge is not slitted.

A traditional approach would be to share some information between the elements that share a face so that the splitting is performed in the same way for the two of them. This is easily done in a scalar context but is non-trivial within a parallel process.

Our proposal attempts to avoid such communication, choosing a splitting pattern exclusively on the basis of locally available data. The idea at the heart of our approach is that such information could be provided by telling how a "uniformly refined" tetrahedra should be collapsed to obtain the desired splitting pattern for all of its faces. This is done by indicating for the edges that are not to be split, the direction toward which the edges of the corresponding uniformly refined element are to be collapsed.

If we assume that a given edge is identified by the GIDs I and J, and that LID(I) and LID(J) give us the local IDs that correspond to the vertices of the element, we will perform for each edge an operation of the type

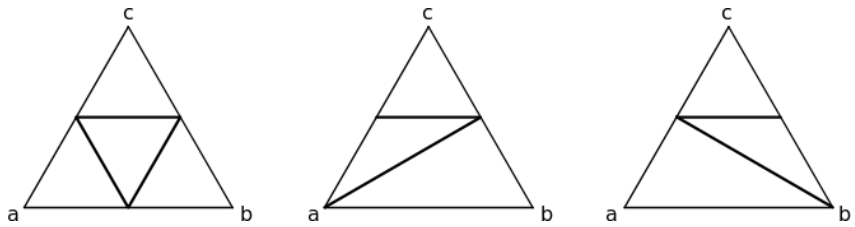


Figure 3: Collapse operations for edge (a,b), from left to right: split edge; edge collapsed towards a; edge collapsed towards b.

```
if(edge is not splitted)
   if(GID(I) < GID(J) )
      collapse_towards = LID(I)
   else
      collapse_towards = LID(J)
else
   collapse_towards = edge_id
```

where the variable "collapse towards" (assigned for each edge of the element) will tell us the direction towards which we should move the node in the center of the edge in order to orient the appropriate collapse. An example of this situation is pictured in Fig(3). Given the face *abc*, assume that edges (b,c) and (c,a) are split. If edge (a,b) is also refined, the triangle is split in four smaller ones. Otherwise, there are two possible outcomes, which can be observed in the figure, and correspond to the situations we have identified as collapsing edge (a,b) towards a and towards b, respectively.

If we apply such algorithm to the example in Fig(2) we will thus associate to the edge (2,3) the tag "2" telling that eventual new edges in the faces that are close to the edge in question will have new edges associated to 2 rather than to 3.

In practice, the overall splitting pattern for the tetrahedra is identified by writing on an auxiliary array of size 6 (number of edges of the element) a tag, that prescribes for each edge which local node should be used in the splitting. A number less than 4, indicates the LID of the node towards which the new edges on the surrounding faces should be oriented. A number exceeding 4 on the other hand, will indicate the LID of a new node to be used in the refinement, corresponding to an unique "edgeId". Table(3) shows the different possibilities for each edge, together with the local numbering we use to identify the edges.

For example the first edge, having edgeindex=0, is identified to the local ids (0,1), the second to (0,2) and so on. We will associate to the first edge either the value of 0 and 1 (the LIDs of its nodes) when the edge is not to be split, or its "edgeId" in the case splitting is required.

As we have 3 possible choices per edge and a total of 6 edges, the total number of possible combinations is of $3^6 = 729$, which need to be considered as potential splitting patterns. The totality of such combinations is considered in the splitting subroutine we provide.

The overall selection process will thus look in pseudo code as:

```
#define an array with the LIDs associated to each edge
edge_id[0] = 4;
edge_id[1] = 5;
edge_id[2] = 6;
edge_id[3] = 7;
edge_id[4] = 8;
edge_id[5] = 9;

edge_counter = 0
for LID1=0 to 2:
  for LID2=1 to 3:
    obtain GID1, GID2 associated to the nodes with LID1, LID2

    if(edge is not splitted)
```

edge index	edge Id	LID1	LID2	possibilities		
0	4	0	1	0	1	4
1	5	0	2	0	2	5
2	6	0	3	0	3	6
3	7	1	2	1	2	7
4	8	1	3	1	3	8
5	9	2	3	2	3	9

Table 1: Definition of the edges of a tetrahedra and possible refinement outcomes.

edge id	LID1	LID2	new LID	GID1	GID2	new GID	reason of choice
4	0	1	0	1	2	1	edge not split and GID1 $<$ GID2
5	0	2	0	1	3	1	edge not split and GID1 $<$ GID2
6	0	3	0	1	4	1	edge not split and GID1 $<$ GID2
7	1	2	1	2	3	2	edge not split and GID1 $<$ GID2
8	1	3	8	2	4	6	insert new node 6 at edge id 8
9	2	3	9	3	4	7	insert new node 7 at edge id 9

Table 2: Top-left tetra - original GIDs 1 2 3 4 - identifier: 0 0 0 1 8 9

```
  if(GID1 < GID2)
     edge_id[edge_counter] = LID1
  else
     edge_id[edge_counter] = LID2
else
   do nothing (already set before the loop)

edge_counter = edge_counter + 1
```

By directly applying such algorithm to the two tetrahedra in Fig(2) we will thus be able to construct Table(2) and Table(3). The final outcome will be to associate to the top left tetrahedra the edge list 0 0 0 1 8 9 and to the right one 0 0 6 2 4 9. This information is enough to univocally select a conformant splitting pattern for the two tetrahedra of interest.

On the other hand, if we consider the non-conformant splitting shown in the bottom right corner and construct its edge list, we will immediately see that a different splitting mode is selected. Table(4) shows the indices obtained for such case, and highlights the point at which the choice is different from the strategy we propose. The final splitting pattern is 0 2 6 2 4 9 which, as expected, does not coincide with the correct one.

The time consuming part of the implementation is certainly the definition of a subroutine that provides the correct splitting pattern once provided the edgeId list. A LGPL licensed C implementation of the splitting strategy proposed can be downloaded freely from the Kratos website [1],[2].

In the attempt of simplifying the interface we provided 3 helper functions, namely “TetrahedraSplitMode”, “Split Tetrahedra” and “TetrahedraGetNewConnectivityGID”.

The user is expected to define an auxiliary vector of size 10, which contains in the first four positions the GIDs of the nodes of the tetrahedra to be split. The positions between 4 and 9 correspond to the edges of the element (ordered as in Table(1)). Their value should be -1 if the edge is not to be split or the GID of the node to be inserted

edge id	LID1	LID2	new LID	GID1	GID2	new GID	reason of choice
4	0	1	0	2	5	2	edge not split and GID1 $<$ GID2
5	0	2	0	2	3	2	edge not split and GID1 $<$ GID2
6	0	3	6	2	4	6	insert new node 6 at edge id 2
7	1	2	2	5	3	3	edge not split and GID1 $>$ GID2
8	1	3	4	5	4	4	edge not split and GID1 $>$ GID2
9	2	3	9	3	4	7	insert new node 7 at edge id 5

Table 3: Top-right tetra - original GIDs 2 5 3 4 - identifier: 0 0 6 2 4 9

edge id	LID1	LID2	new LID	GID1	GID2	new GID	follows convention?
4	0	1	0	2	5	2	OK
5	0	2	2	2	3	3	NOT FOLLOWING CONVENTION!
6	0	3	6	2	4	6	OK
7	1	2	2	5	3	3	OK
8	1	3	4	5	4	4	OK
9	2	3	9	3	4	7	OK

Table 4: Bottom-right tetra - original GIDs 2 5 3 4 - identifier: 0 2 6 2 4 9

	first step		second step	
# threads	time (sec)	speedup	time (sec)	speedup
4	21.16	1.0	170.74	1.0
8	10.57	2.0	88.09	1.9
16	5.71	3.7	48.99	3.5
32	3.25	6.5	24.99	6.8

Table 5: Time required to refine a 1 million element mesh twice.

if splitting is required.

The function "TetrahedraSplitMode" takes as input such auxiliary vector and returns a second work vector which assigns a local id to the edges as described in the paragraph above.

The output of this function is used as input for "Split Tetrahedra" which performs the splitting and returns an array with the LIDs of the new tetras. A flag is returned to identify that a new central node is needed.

Finally the function "TetrahedraGetNewConnectivityGID" simplifies the creation of the mesh.

The documentation attached to the file provides a detailed example of usage.

4 Parallel Benchmark

To test the parallel efficiency of the element splitting strategy presented in the previous pages, a simple homogeneous refinement example is executed. Consider a cubic domain identified by its corners $(-1,-1,-1)$ and $(1,1,1)$, initially meshed using slightly over one million tetrahedral elements. Such domain is refined homogeneously in two passes, first splitting all of its elements to obtain around 8 million elements and again for a total of 64 million elements.

Computations where performed using up to three blades containing two six-core Intel Xeon E5645 CPU ($2.40\,GHz$, $48\,Gb$ RAM) each, connected using Infiniband. The time required to perform this operation using an increasing amount of processors is recorded in Table(5) and presented in graphical form in Fig(4). The results show that the refinement algorithm exhibits a good parallel performance, as expected.

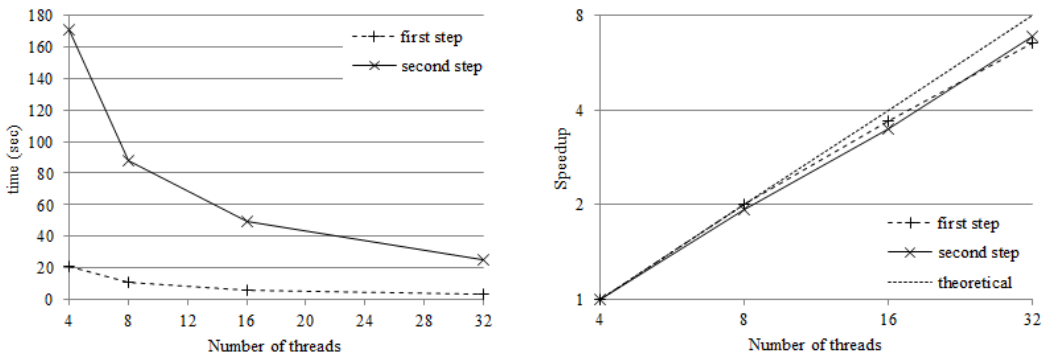


Figure 4: Wall time and speedup for the parallel performance benchmark.

5 A simple subscale-based error indicator

In the present section an error estimation technique for incompressible flow problems will be introduced. This technique is not directly tied to the refinement strategy described in the previous sections, but is presented as a practical example of a criterion to drive the adaptive refinement algorithm that can be used in application examples.

This refinement technique is closely related to the variational multiscale method for the stabilization of the Navier-Stokes equations, introduced in [22] and [23]. To provide a context for the error estimator we will proceed to briefly describe the problem.

5.1 Variational multiscale formulation of the incompressible Navier-Stokes equations

The starting point of the formulation are the incompressible Navier-Stokes equations

$$\partial_t \mathbf{u} + \mathbf{u} \cdot \nabla \mathbf{u} - \nabla \cdot (2\nu \nabla^{\mathbf{s}} \mathbf{u}) + \nabla p = \mathbf{f} \tag{1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2}$$

where $\mathbf{u}$ and p represent velocity and pressure, ν is the fluid's kinematic viscosity, $\mathbf{f}$ is the vector of external forces and $\nabla^s \mathbf{u} := \frac{1}{2}\nabla \mathbf{u} + \frac{1}{2}(\nabla \mathbf{u})^T$ is the symmetric part of the velocity gradient.

As Eq (1) contains a non-linear convective term, $\mathbf{u} \cdot \nabla \mathbf{u}$, we will first linearize it using Picard's method. Using the index i to denote a result from the previous iteration, we obtain the linearized momentum equation

$$\partial_t \mathbf{u} + \mathbf{u}^i \cdot \nabla \mathbf{u} - \nabla \cdot (2\nu \nabla^{\mathbf{s}} \mathbf{u}) + \nabla p = \mathbf{f} \tag{3}$$

If f and g are functions such that their product fg is integrable in the domain Ω, the following standard compact notation can be introduced

$$\int_\Omega fg \, \mathrm{d}\Omega = (f,g)_\Omega$$

similarly, Ω_e will be used to to denote integrals over a single element.

By multiplying Eq(3) and Eq(2) by test functions $\mathbf{v}$, q and integrating over the fluid domain Ω the Galerkin weak form of the Navier-Stokes equations is obtained.

$$(\mathbf{v}, \partial_t \mathbf{u})_\Omega + (\mathbf{v}, \mathbf{u}^i \cdot \nabla \mathbf{u})_\Omega + 2\nu(\nabla^s \mathbf{v}, \nabla^s \mathbf{u})_\Omega - (\nabla \cdot \mathbf{v}, p)_\Omega = (\mathbf{v}, \mathbf{f})_\Omega \tag{4}$$

$$(q, \nabla \cdot \mathbf{u})_\Omega = 0 \tag{5}$$

It is well known that the numerical solution of the Navier-Stokes equations runs into numerical instabilities due to the incompressibility constraint, as well as due to the convective term in convection-dominated flows. A series of stabilization techniques have been developed over the last decades to overcome these numerical instabilities, e.g. SUPG [25], GLS [24] or FIC [37, 36, 35]. One of them is the variational multiscale method, which we will use here, based on the division of the solution on large scale and small scale parts.

$$\mathbf{u} = \mathbf{u}_h + \tilde{\mathbf{u}} \qquad p = p_h + \tilde{p}$$

The large scale part of the solution, $\mathbf{u}_h$, p_h, represents the component of the exact solution that can be reproduced using a given spatial discretization, while the small scale part, or subscale, is the difference between the exact solution and the result of the discretized problem. By introducing the scale separation for the problem variables and test functions on Eq(4) and Eq(5), two different equations are obtained. Omitting the details on their derivation (which can be found for example [12]), and neglecting some terms involving integrals over element boundaries or second derivatives of the test functions, the large scale equations read

$$\begin{aligned}(\mathbf{v}_h, \partial_t \mathbf{u}_h)_\Omega + (\mathbf{v}_h, \mathbf{u}_h^i \cdot \nabla \mathbf{u}_h)_\Omega + 2\nu(\nabla^s \mathbf{v}_h, \nabla^s \mathbf{u}_h)_\Omega + (\nabla \cdot \mathbf{v}_h, p_h)_\Omega \\ + \sum_e (\mathbf{u}_h^i \cdot \nabla \mathbf{v}_h, \tilde{\mathbf{u}})_{\Omega_e} + \sum_e (\nabla \cdot \mathbf{v}_h, \tilde{p})_{\Omega_e} = (\mathbf{v}_h, \mathbf{f})_\Omega\end{aligned} \tag{6}$$

$$(\mathbf{v}_h, \nabla \cdot \tilde{\mathbf{u}})_\Omega + \sum_e (\nabla q_h, \tilde{\mathbf{u}})_{\Omega_e} = 0 \tag{7}$$

while the small scales are driven by

$$(\tilde{\mathbf{v}}, \partial_t \tilde{\mathbf{u}})_\Omega + (\tilde{\mathbf{v}}, \mathbf{u}_h^i \cdot \nabla \tilde{\mathbf{u}})_\Omega - 2\nu(\tilde{\mathbf{v}}, \nabla \cdot (\nabla^s \tilde{\mathbf{u}}))_\Omega - (\nabla \cdot \tilde{\mathbf{v}}, \tilde{p})_\Omega =$$
$$(\tilde{\mathbf{v}}, \mathbf{f})_\Omega - (\tilde{\mathbf{v}}, \partial_t \mathbf{u}_h)_\Omega - (\tilde{\mathbf{v}}, \mathbf{u}_h^i \cdot \nabla \mathbf{u}_h)_\Omega + 2\nu(\tilde{\mathbf{v}}, \nabla \cdot (\nabla^s \mathbf{u}_h))_\Omega - (\nabla \cdot \tilde{\mathbf{v}}, p_h)_\Omega \quad (8)$$
$$(\tilde{q}, \nabla \cdot \tilde{\mathbf{u}})_\Omega = -(\tilde{q}, \nabla \cdot \mathbf{u}_h)_\Omega \quad (9)$$

The aim of the variational multiscale method is to solve the large scale equations Eq(6) and Eq(7) by modeling the effect the small scale terms $\tilde{\mathbf{u}}$, $\tilde{p}$ have on them. The modeling terms that will be introduced are motivated by the small scale functions Eq(8) and Eq(9), using an argument based on the small scale Green's function.

Observe that the small scale equations should be verified for all functions $\tilde{\mathbf{v}}$ and $\tilde{q}$ in the spaces of velocity and pressure subscales respectively. As such, they can be considered as equations imposed over the L^2-*projection* of a differential equation onto the space of small scales. Using this observation, Eq(8) and Eq(9) can be recast in differential form as

$$\partial_t \tilde{\mathbf{u}} + \mathbf{u}_h^i \cdot \nabla \tilde{\mathbf{u}} - \nabla \cdot (2\nu \nabla^s \tilde{\mathbf{u}}) + \nabla \tilde{p} = \mathbf{r}_u(\mathbf{u}_h, p_h) - \xi_h \quad (10)$$
$$\nabla \cdot \tilde{\mathbf{u}} = r_p(\mathbf{u}_h) - \delta_h \quad (11)$$

where $\mathbf{r}_u$ and r_p represent the residuals of the momentum and mass equations applied to the large scale variables, defined as

$$\mathbf{r}_u(\mathbf{u}_h, p_h) = \mathbf{f} - \partial_t \mathbf{u}_h - \mathbf{u}_h^i \cdot \nabla \mathbf{u}_h + \nabla \cdot (2\nu \nabla^s \mathbf{u}_h) - \nabla p_h \quad (12)$$
$$r_p(\mathbf{u}_h) = -\nabla \cdot \mathbf{u}_h \quad (13)$$

and ξ_h (and δ_h) are such that, once added to the momentum (or mass) residuals, the sum belongs to velocity (or pressure) small scale space.

Unfortunately, the space containing the small scale solutions is infinite-dimensional, unlike the large scale one, which is a finite element space, and must be approximated by a finite-dimensional space. The choice of approximation for the small scale space determines the definition for the projection terms ξ_h, δ_h. One common choice is just assuming them to be zero, which is what was done in the original papers on the variational multiscale method and is denominated as algebraic subgrid scales (ASGS) by [11], a notation that will be followed here. An alternative is considering the space of subscales L^2 orthogonal to the space of large scales. This option was presented in [10] and [12], where it is called orthogonal subscales (OSS), and results in a method very similar to projection schemes. In this case, the projection terms ξ_h, δ_h are defined as the projection of the respective residual onto the large scales, which ensures that the right hand side in Eq(10) and Eq(11) is orthogonal to the space of large scales:

$$\xi_h = \Pi_{V_h}\left(\mathbf{f} - \partial_t \mathbf{u}_h - \mathbf{u}_h^i \cdot \nabla \mathbf{u}_h + \nabla \cdot (2\nu \nabla^s \mathbf{u}_h) - \nabla p_h\right) \quad (14)$$
$$\delta_h = \Pi_{Q_h}(-\nabla \cdot \mathbf{u}_h) \quad (15)$$

5.2 The small-scale Green's function

Although Eq(8) and Eq(9) define the subscale velocity and pressure, they are not usually solved in practice. Instead, the subscale terms that appear in the large scale Eq(6) and Eq(7) are approximated using an argument based on the Green's function associated to Eq(10) and Eq(11). This procedure will be introduced here, but the interested reader is directed to the foundational papers on variational multiscale methods, such as [23], for a more complete presentation.

Given a problem such as

$$\mathcal{L}u = f \quad \text{in } \Omega$$
$$u = 0 \quad \text{on } \partial\Omega$$

the Green's function associated to $\mathcal{L}$ is defined as a function $G : \Omega \times \Omega \to \mathbb{R}$ such that

$$u(x) = \int_\Omega G(x,y) f(y)\, \mathrm{d}\Omega \quad \forall x \in \Omega \quad (16)$$

where the integral has to be understood in a distributional sense.

Applying this concept to the equations that define the small scales, Eq(10) and Eq(11), there exist $\mathbf{G}_u$ and G_p, called small scale Green's function for the velocity and for the pressure respectively, such that

$$\tilde{\mathbf{u}} = \int_\Omega \mathbf{G}_u \left(\mathbf{r}_u - \xi_h\right) \mathrm{d}\Omega$$
$$\tilde{p} = \int_\Omega G_p \left(r_p - \delta_h\right) \mathrm{d}\Omega$$

Note that the small scale Green's functions defined by these equations are global. In the context of a spatial discretization, the Green's function for the subscales can be defined locally for each element, provided that the subscales are assumed to be zero on element boundaries, which is a common assumption in stabilized methods. In this way, local small scale Green's functions can be defined, using a single element as integration domain.

In practice, the local small scale Green's function is not calculated. Instead, an algebraic approximation is defined as

$$\tilde{\mathbf{u}} \approx \tau_1 \left(\mathbf{r}_u - \xi_h\right) \qquad \tilde{p} \approx \tau_2 \left(r_p - \delta_h\right) \tag{17}$$

where the second order tensor τ_1 and the scalar τ_2 are called stabilization parameters, have dimensions of time and will have to be determined.

In the present work the stabilization parameters are implemented, according to the definitions in [11], as

$$\tau_1 = \left(\frac{c_1 \nu}{h^2} + \frac{c_2 \left|\mathbf{u}^i\right|}{h} \right)^{-1} \mathbf{I} \tag{18}$$

$$\tau_2 = \frac{h^2}{c_1 \tau_1} \tag{19}$$

where $\mathbf{I}$ is the second order identity tensor, h is a characteristic length of the element and the parameters take the values $c_1 = 4$, $c_2 = 2$ for linear elements.

Using Eq(17), the concept of the small scale Green's function provides a closure for the large scale Eq(6) and Eq(7), motivating an approximation to the subgrid terms $\tilde{\mathbf{u}}$, $\tilde{p}$ that avoids the necessity to solve additional equations.

5.3 Subscale-based error estimation

From the point of view of the error estimation, as pointed out in [23] and [18] the approximation to the subscales is ultimately proportional to the elemental residual of the large-scale equations. As such, it is a natural choice as an error estimator within the framework of variational mulstiscale methods. An approach based on this idea is analyzed in [18], where the performance of a subscale based error estimator (using the formulation defined as ASGS above) is studied.

The error estimator used in the examples presented in this document is defined as

$$\varepsilon = \frac{\left\| \tau_1 \left(\mathbf{r}_u - \xi_h\right) \right\|}{\left\| \mathbf{u}_h^{avg} \right\|} \tag{20}$$

where $\|\cdot\|$ denotes the L^2 norm and $\mathbf{u}_h^{avg}$ is an average large scale velocity on the domain. This error estimator is evaluated on the element centers by the AMR routine and, if it is found to be larger than a predefined tolerance, the element is refined.

6 Incompressible flow examples

To conclude we present some examples of application of the refinement algorithm with the error estimation technique described on the previous section. The first two examples have been run using four processes on a desktop computer, while the last 3D case was run using 32 cores of the *Atlante* cluster. The *Atlante* cluster, located at the

Insituto Ténico de Canarias, comprises 84 JS21 computation nodes (blades) each with two dual-core PPC970 processors at 2.5 GHz. Each blade has 8 GB RAM. Parallel communications are performed over a Myrinet network.

6.1 An illustrative small-scale example

To show a simple example application of the procedure outlined in this document, the results obtained for two-dimensional incompressible flow around a cylinder are presented here. This problem, taken from [12], has been solved using OSS stabilization. Let D be the diameter of a cylinder centered on the origin of the domain $[-4D, 12D] \times [-4D, 4D]$. An inlet condition is imposed on the left side of the domain, with a velocity such that the Reynolds number computed with D is $\mathrm{Re} = 100$.

Starting from a mesh of 3984 triangular linear elements, the flow has been simulated during 60 seconds of flow, using a time step of 0.1 seconds. After an initial waiting phase, the AMR is started, using the subscale based estimator introduced in the previous pages, checking the error every 20 solution steps for a total of 40 refinement passes (although the refinement is limited to 3 passes over the same area and a minimum element size is imposed, to preserve mesh quality and prevent excessive refinement on localized areas).

The simulation was run, for test purposes, using 8 processes and distributed memory in a cluster at CIMNE. At the end of the simulation, a final mesh containing 11666 elements was obtained, which is reproduced in Fig(5). The mesh was distributed over the eight domain partitions as shown in Table 6. It can be seen that the refined area coincides with the region where vortices develop, which is what would be expected.

Process	0	1	2	3	4	5	6	7	Total
Initial	496	510	501	493	494	490	505	495	3984
Final	1056	1871	836	740	1884	1694	1542	2043	11666

Table 6: Initial and final number of elements on each process.

Observing the results in Table 6, it is evident that the mesh refinement is not homogeneous between the domains. This is reasonable, in the sense that a successful refinement strategy should concentrate the elements in critical areas, but not desirable from the point of view of parallel efficiency, and highlights the need to couple the refinement strategy with a load balancing algorithm when it is used on a parallel environment.

6.2 Rectangular cylinder

Another example where the algorithm presented in this document was tested is the flow over a rectangular cylinder ($20 \times 1\ m$) with rounded corners (radius $0.3\ m$). The fluid properties were density $\rho = 1.225\ Kg/m^3$ and kinematic viscosity $\nu = 1.46 \times 10^{-5}$. The flow was defined by an incoming velocity of $20\ m/s$ in the direction of the long edges of the rectangle. After an initial waiting phase while the solution develops, the AMR algorithm is used every 20 solution steps to refine the mesh, again limiting the maximum amount of refinement passes over the same area, as well as the minimum element size. Unlike the previous case, the time step is now fixed at run time to maintain the elemental Courant-Friederichs-Levy number over 10. Another difference is that, in this case, after every refinement step some edge swapping is performed to improve mesh quality. The results of the simulation can be seen in Fig(6), where again it is appreciated that the refinement is concentrated on the wake of the body, as expected.

The initial mesh contains 37.829 nodes and 73.290 elements, which are increased to a total of 80.726 nodes and 159.084 elements after 1.6 seconds of simulation.

6.3 Flow around the Silsoe cube

As a final example, a simulation of a $3D$ incompressible flow problem will be presented. The geometry for this benchmark has been taken from [40], where measurements of the wind flow around a six meter cube constructed at Silsoe Research Institute are presented.

Our simulation represents the flow around the cube when the incoming wind is perpendicular to one of its faces, and simulates a tetrahedral domain $108\,m$ long on the direction of the flow, $48\,m$ long in the perpendicular direction and $30\,m$ high. In the inflow boundary, placed $60\,m$ before the center of the cube, the following logarithmic velocity profile is imposed:

$$u_z = \frac{u_*}{\kappa} \log\left(\frac{z u_*}{\nu}\right) + B \tag{21}$$

where u_z is the longitudinal velocity at height z, $u_* = 0.272$ is the friction velocity, $\nu = 1.51 \times 10^{-5} m^2/s$ the kinematic viscosity of air, $\kappa = 0.41$ Von Kármán's constant and $B = 5.2$. The air density is considered to be $\rho = 1.225 Kg/m^3$. A total time of 6 seconds has been simulated, using a time step of 0.1 seconds.

The refinement algorithm has been run initially after 20 simulation steps and every 10 steps after that point, for a total of five refinement passes. Refinement has been performed for all elements where the error estimator evaluated a subscale velocity larger than a 5% of the average large-scale velocity, limited to two refinements over a single original element.

The results of the simulation at time $1\,s$, with the original mesh, and time $6\,s$, with the refined mesh, are presented in Fig(7). The domain was meshed using 1.6 million elements initially and, after five refinement steps, the algorithm produced a domain with a total of 5.3 million elements. Note how the refined elements are concentrated near the main features of the flow, and correctly catches the formation of the horseshoe vortex on the front of the cube.

7 Concluding remarks

This document presents a mesh refinement algorithm designed with its use in a parallel environment in mind. The solution proposed is relatively simple to implement, as it is based on structures that are commonly provided by linear algebra libraries, such as distributed sparse matrices. that will be already available in most parallel finite element codes.

The procedure presented has three basic components. The first of them is an element-driven global splitting algorithm, that identifies the elements that must be subdivided and communicates this information to all processes. This component relies on an error estimation strategy, which identifies the areas where mesh resolution is insufficient. The third component of the algorithm is a local refinement procedure, which subdivides the existing elements in a way that ensures that they will be conformant with their neighbors.

Error estimation is dependent on the physical formulation of the problem that is solved. This paper presents one choice of error estimator, specific to incompressible flow problems, which has been used in some simple examples. Obviously, this is just one possible estimator, and other options will be more desirable for other problems, but the main refinement algorithm is not problem-dependent, and can be used to refine any arbitrary subset of elements in the domain.

An important question that has not been addressed in this document is that, when the finite element mesh is adaptively refined, new elements will be created in localized areas of the domain and, as a result, the number of elements in each parallel subdomain will change. A crucial line of future work will be defining a strategy to preserve load balance when this happens. Another line of improvement is to provide a mesh coarsening strategy, to reverse the refinements performed if the error is found to be sufficiently small in later time steps.

A second important line of future research is the improvement of the mesh quality of the refined meshes. For example in the case of stretched or badly shaped elements it is potentially interesting to employ the ideas of the "longest edge-refinement" techniques.

Acknowledgements

The authors wish to acknowledge the support of the Spanish *Ministerio de Ciencia e Innovación* through the E-DAMS project and the European Commission through the Realtime project. In addition, the authors thankfully acknowledge the computer resources, technical expertise and assistance provided by the *Red Española de Supercomputación*. Nelson Lafontaine thanks to MAEC-AECID scholarships for financial support given. J. Cotela gratefully acknowledges the support of the Spanish *Ministerio de Educación* through a doctoral grant in the FPU program and the *Col·legi d'Enginyers de Camins, Canals i Ports*.

References

[1] Kratos Multiphysics. http://kratos.cimne.upc.es/kratoswiki/index.php/Main _Page.

[2] Tetrahedra split routine. http://kratos.cimne.upc.edu/trac/browser/kratos/kratos/utilities/split _tetrahedra.c.

[3] I. Babuscka and A. Miller. A posteriori error estimates and adaptative techniques for the finite element method. *Technical Report BN-968, Institute for Physical Science and Technology, Univerisity of Maryland*, 1981.

[4] Mehmet Balman. Parallel tetrahedral mesh refinement. Master Thesis. Bogazici University, 2000, Jan 2006.

[5] Randolph E. Bank and Andrew H. Sherman Alan Weiser. Some Refinement Algorithms And Data Structures For Regular Local Mesh Refinement. 1983.

[6] Eberhard Bansch. An adaptive finite-element strategy for the three-dimensional time-dependent Navier-Stokes equations. *Journal of Computational and Applied Mathematics*, 36(1):3–28, 1991.

[7] William J. Barry, Mark T. Jones, and Paul E. Plassmann. Parallel adaptive mesh refinement techniques for plasticity problems. *Adv. Eng. Softw.*, 29:217–225, April 1998.

[8] Mark E. Botkin and Hui-Ping Wang. An adaptive mesh refinement of quadrilateral finite element meshes based upon an a posteriori error estimation of quantities of interest: modal response. *Eng. Comput. (Lond.)*, pages 38–44, 2004.

[9] K. C. Chellamuthu and N. Ida. Algorithms and data structures for 2D and 3D adaptive finite element mesh refinement. *Finite Elements in Analysis and Design*, 17(3):205–229, 1994.

[10] Ramon Codina. Stabilization of incompressibility and convection through orthogonal sub-scales in finite element methods. *Computer Methods in Applied Mechanics and Engineering*, 190(13-14):1579–1599, 2000.

[11] Ramon Codina. A stabilized finite element method for generalized stationary incompressible flows. *Computer Methods in Applied Mechanics and Engineering*, 190(20-21):2681–2706, 2001.

[12] Ramon Codina. Stabilized finite element approximation of transient incompressible flows using orthogonal subscales. *Computer Methods in Applied Mechanics and Engineering*, 191(39-40):4295–4321, 2002.

[13] H. L. De Cougny and M. S. Shephard. Parallel refinement and coarsening of tetrahedral meshes. *International Journal for Numerical Methods in Engineering*, 46(7):1101–1125, 1999.

[14] Pooyan Dadvand, Riccardo Rossi, and Eugenio Oñate. An Object-oriented Environment for Developing Finite Element Codes for Multi-disciplinary Applications. *Archives of Computational Methods in Engineering*, 17:253–297, 2010.

[15] Thomas Gratsch and Klaus-Jurgen Bathe. A posteriori error estimation techniques in practical finite element analysis. *Computers & Structures*, 83(4-5):235–265, 2005.

[16] Roberto Grosso, Christoph Lürig, and Thomas Ertl. The multilevel finite element method for adaptive mesh optimization and visualization of volume data. In *Proceedings of the 8th conference on Visualization '97*, VIS '97, pages 387–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[17] Hertel R.; Kronmuller H. Adaptive Finite Element Mesh Refinement Techniques in Three-Dimensional Micromagnetic Modeling. *IEEE Transactions on Magnetics*, 34(6):3922–3930, Nov 1998.

[18] Guillermo Hauke, Mohamed H. Doweidar, and Mario Miana. The multiscale approach to error estimation and adaptivity. *Computer Methods in Applied Mechanics and Engineering*, 195(13-16):1573–1593, 2006. A Tribute to Thomas J.R. Hughes on the Occasion of his 60th Birthday.

[19] Alejandro Díaz Morcillo; Luis Nuño; Juan V. Balbastre; David Sánchez Hernández;. Adaptative mesh refinement in electromagnetic problems. Oct 2000.

[20] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[21] N. Jansson; J. Hoffman and J. Jansson. Parallel Adaptive FEM CFD. *Technical Report KTH-CTL-4008, Computational Technology Laboratory*, 2010.

[22] Thomas J. R. Hughes. Multiscale phenomena: Green's functions, the Dirichlet-to-Neumann formulation, subgrid scale models, bubbles and the origins of stabilized methods. *Computer Methods in Applied Mechanics and Engineering*, 127(1-4):387–401, 1995.

[23] Thomas J. R. Hughes, Gonzalo R. Feijóo, Luca Mazzei, and Jean-Baptiste Quincy. The variational multiscale method–a paradigm for computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 166(1-2):3–24, 1998. Advances in Stabilized Methods in Computational Mechanics.

[24] Thomas J. R. Hughes, Leopoldo P. Franca, and Gregory M. Hulbert. A new finite element formulation for computational fluid dynamics: VIII. The galerkin/least-squares method for advective-diffusive equations. *Computer Methods in Applied Mechanics and Engineering*, 73(2):173–189, 1989.

[25] Thomas J. R. Hughes and Michel Mallet. A new finite element formulation for computational fluid dynamics: III. The generalized streamline operator for multidimensional advective-diffusive systems. *Computer Methods in Applied Mechanics and Engineering*, 58(3):305–328, 1986.

[26] Raizer A. ; Meunier G.; Coulomb J.L. An approach for automatic adaptive mesh refinement in finite element computation of magnetic fields. *IEEE Transactions on Magnetics*, 25(4):2965–2967, Jul 1989.

[27] Mark T. Jonesa; and Paul E. Plassmannb. Adaptive refinement of unstructured finite-element meshes. *Finite Elements in Analysis and Design*, 25(1-2):41–46, March 1997.

[28] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers*, 22(3–4):237–254, 2006.

[29] Igor Kossaczky. A recursive approach to local mesh refinement in two and three dimensions. *Journal of Computational and Applied Mathematics*, 55(3):275–288, 1994.

[30] Y.-Y. Lian, K.-H. Hsu, Y.-L. Shao, Y.-M. Lee, Y.-W. Jeng, and J.-S. Wu. Parallel adaptive mesh-refining scheme on a three-dimensional unstructured tetrahedral mesh and its applications. *Computer Physics Communications*, pages 721–737, 2006.

[31] Y.-Y. Lian, K.-H. Hsu, Y.-L. Shao, Y.-M. Lee, Y.-W. Jeng, and J.-S. Wu. Parallel adaptive mesh-refining scheme on a three-dimensional unstructured tetrahedral mesh and its applications. *Computer Physics Communications*, 175(11-12):721–737, 2006.

[32] Anwei Liu and Barry Joe. Quality Local Refinement of Tetrahedral Meshes Based on Bisection. 16(6):1269–1291, 1995.

[33] T.W.; Field D.A Nehl. Adaptive refinement of first order tetrahedral meshes for magnetostatics using local Delaunay subdivisions. *IEEE Transactions on Magnetics*, 27(5):4193–4196, Sep 1991.

[34] Ríos Rodriguez; Gustavo A.; Storti M.A; Nigro N.M. Adaptive Refinement of Unstructured Finite Element Meshes for Compressible Flowss. *Mecánica Computacional*, 27(5):1283–1295, Nov 2009.

[35] E. Oñate, A. Valls, and J. García. Computation of turbulent flows using a finite calculus–finite element formulation. *International Journal for Numerical Methods in Fluids*, 54(6-8):609–637, 2007.

[36] Eugenio Oñate, Juan Miquel, and Guillermo Hauke. Stabilized formulation for the advection–diffusion–absorption equation using finite calculus and linear finite elements. *Computer Methods in Applied Mechanics and Engineering*, 195(33-36):3926–3946, 2006.

[37] Eugenio Oñate, Francisco Zárate, and Sergio R. Idelsohn. Finite element formulation for convective–diffusive problems with sharp gradients using finite calculus. *Computer Methods in Applied Mechanics and Engineering*, 195(13-16):1793–1825, 2006. A Tribute to Thomas J.R. Hughes on the Occasion of his 60th Birthday.

[38] Boluan Pang. A tetrahedral refinement algorithm for adaptive finite element methods in electromagnetics. Master Tesis. McGill University, Montreal, Aug 2009.

[39] Philippe P. Pebay and David C. Thompson. Parallel Mesh Refinement Without Communication. *Proeceedings 13th International Meshing Roundtable, Williamsbourg USA*, 2004.

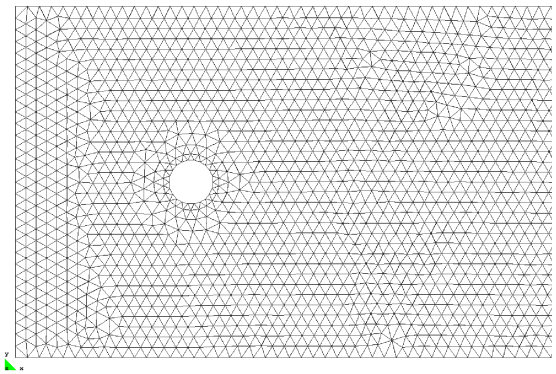
[40] P.J. Richards, R.P. Hoxey, and L.J. Short. Wind pressures on a 6m cube. *Journal of Wind Engineering and Industrial Aerodynamics*, 89(14–15):1553 – 1564, 2001.

[41] M.-C. Rivara. Mesh refinement processes based on the generalized bisection of simplices. *SIAM J. Numer Anal*, 21:604–613, 1984.

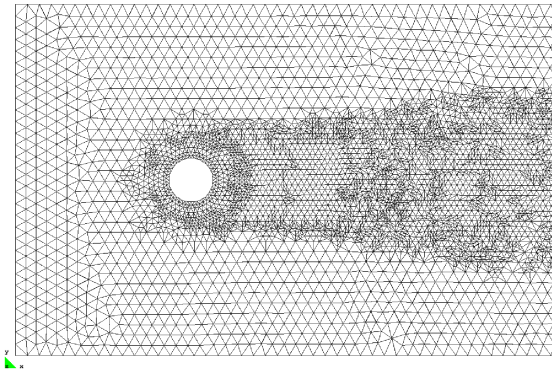
[42] Maria-Cecilia Rivara. Mesh Refinement Processes Based on the Generalized Bisection of Simplices. *SIAM Journal on Numerical Analysis*, 21(3):pp. 604–613, 1984.

[43] E. G. Sewell. Automatic generation of triangulation for piecewise polynomial approximation. Ph. D. Thesis Purdue Univ., West Lafayette, Jan 1972.

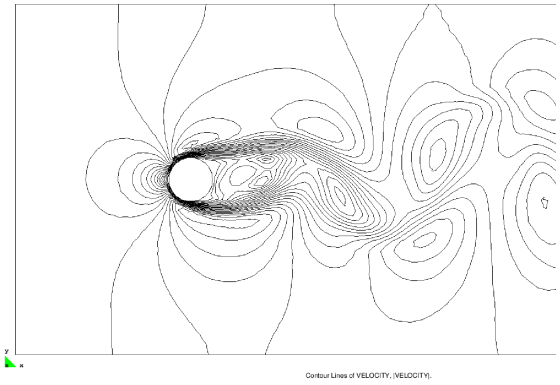
[44] L. Bo Zhang. A Parallel Algorithm for Adaptive Local Refinement of Tetrahedral Meshes Using Bisection. *Tech. Rep. Preprint ICM-05-09, Institute of Computational Mathematics and Scientific/Engineering Computing.*, 2005.



(a)

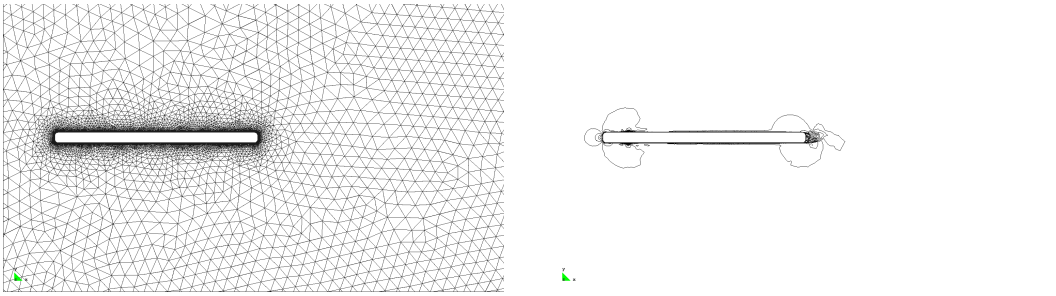


(b)

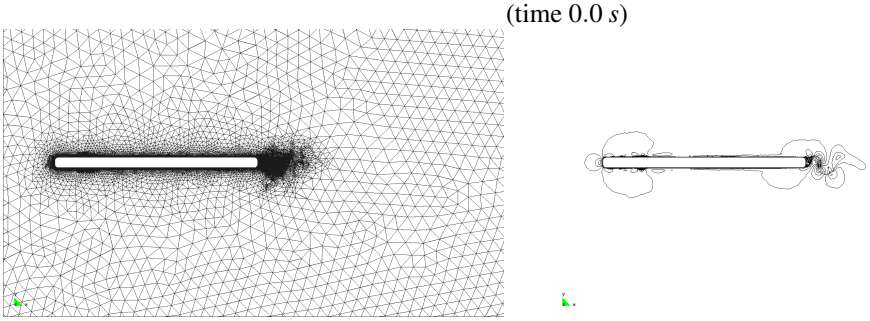


(c)

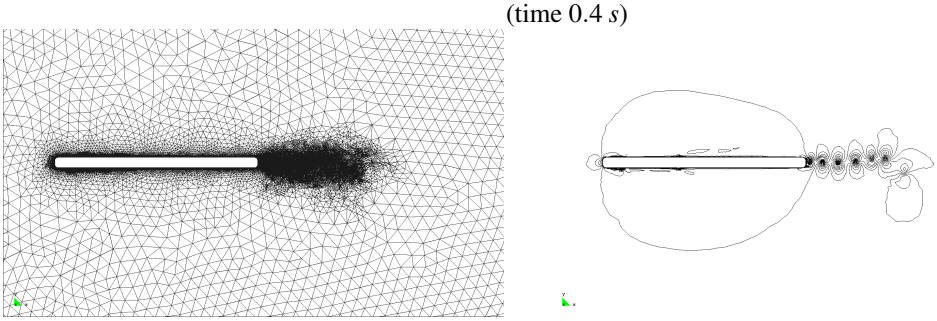
Figure 5: Initial mesh (a), refined mesh and (b) final velocity contours (c) for the cylinder example.



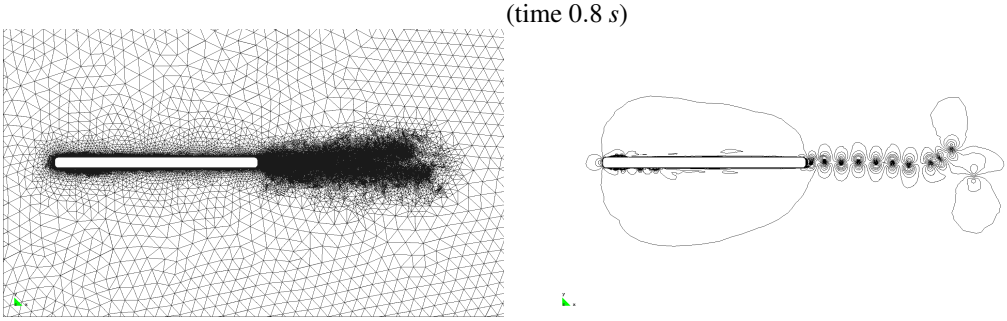
(time 0.0 *s*)



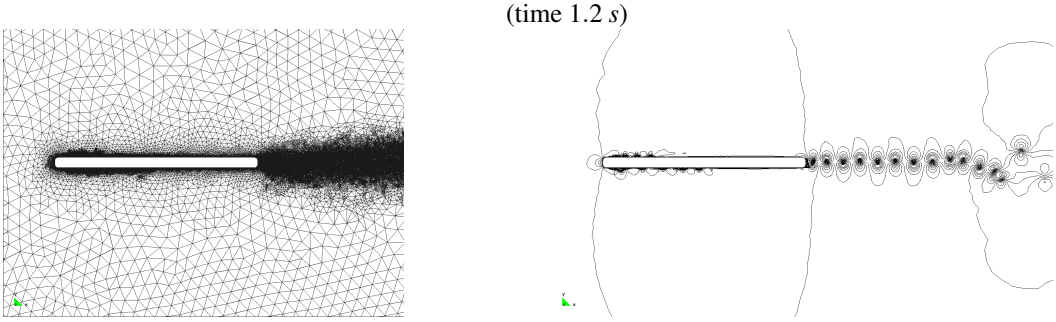
(time 0.4 *s*)



(time 0.8 *s*)

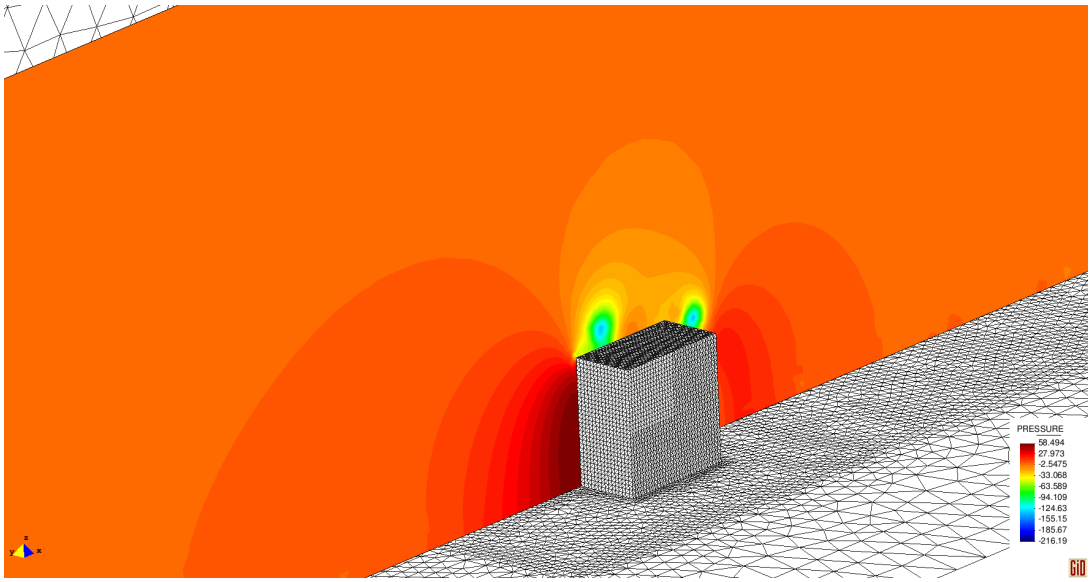


(time 1.2 *s*)

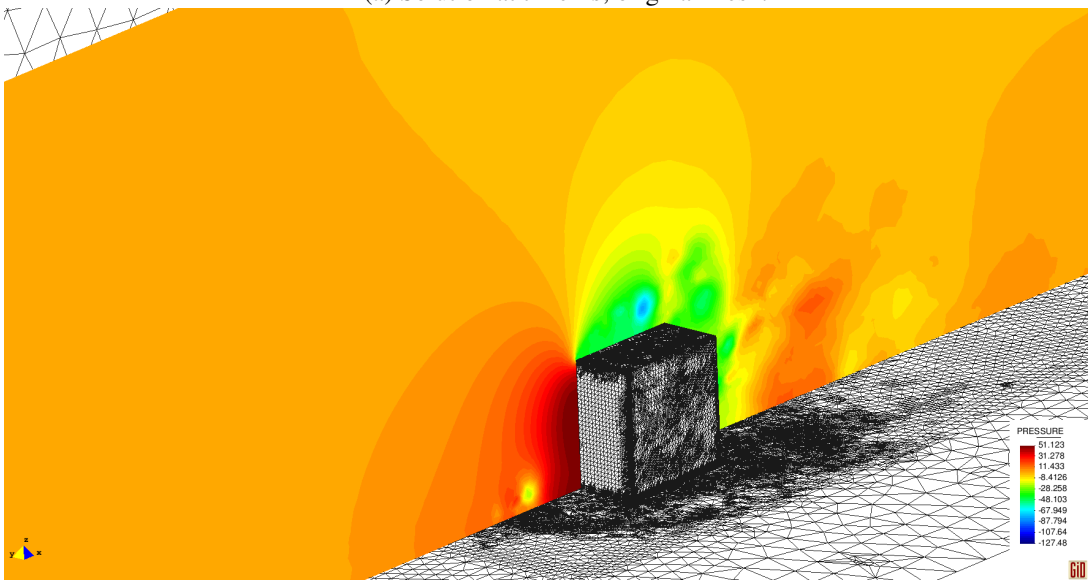


(time 1.6 *s*)

Figure 6: Velocity solution and refined mesh for the rectangle example.



(a) Solution at time $1\,s$, original mesh.



(b) Solution at time $6\,s$, refined mesh.

Figure 7: Pressure results on the midplane for the flow over the Silsoe cube.